ROBUST CAMERA ARCHITECTURE TO SUPPORT SPECIALIZED CAMERA SYSTEMS

BY

DEAN EDWARD GLAZESKI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Narendra Ahuja

ABSTRACT

Live camera applications have historically kept to a simple view of cameras as a single-stream of images that can be displayed in a single, two-dimensional view. Because of the inherent limitation in doing this, existing frameworks are not flexible enough to enable custom camera systems involving multiple image streams or cameras that have special display needs. This has left a need for a new architecture that can support those camera systems that are held back by the classic view of a camera. This architecture will illustrate how it can be extended through plug-ins to support any kind of camera configuration and display available today. It will also show that it can accomplish this while still providing at least 25 FPS performance on a 5 MP image stream. In addition, attempts are made to show how multi-stream image synchronization can be achieved as an extension of the architecture in order to support the primary camera target, a HemView camera system. This thesis also serves as documentation for the architecture that was created and how each of the provided plug-ins were built to fit into the system.

To my parents, for their love and support

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
FIFO	First In First Out
FPS	Frames Per Second
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
MP	Megapixel
PTZ	Pan Tilt Zoom
SDK	Software Development Kit

CHAPTER 1

PROBLEM STATEMENT

The majority of digital cameras in use today consist of a single sensor. This makes it very easy to abstract these cameras into just a stream of discreet images coming from a single source. This has yielded many interesting software products from high zoom PTZ cameras to 360° devices that make use of special fisheye lenses. This simple idea has spawned many software products that are capable of displaying and configuring these various systems. These software products have one glaring problem that stems from the simplistic view of cameras today.

Existing video software solutions think of cameras as a single stream of information. New cameras currently in development are starting to make use of multiple video channels to create a wider field of view or to image a whole sphere in a single go. Current video software is not capable of working with these systems and a generic approach is needed to address current and future camera designs that take advantage of multiple streams. As an example, the HemView camera is one such system that uses multiple images to produce a single image [1]. Depth cameras also serve as an example system that are becoming more and more pervasive and can take advantage of advancements in camera software architecture.

There are several target goals for this experimental architecture. The first is to provide a robust camera architecture that can support at least 25 frames per second with an image sensor of at least 5 MP. The second goal is to create a HemView infrastructure that allows for easy adjustment of the number and type of cameras that make up the array. The third goal is to generate a full HemView display with proper camera synchronization. The final goal is to support many different types of cameras from firewire to IP cameras to make an easy-to-use and robust camera application.

This document will address the different aspects of the experimental architecture. The first chapter addresses the existing solutions and points out

how this architecture will differentiate itself. The next chapter discusses the different dependencies used by the application to build the final architecture. The third chapter details the various pieces that make up the core architecture. Following that chapter, the plug-in infrastructure is discussed along with some sample plug-ins that illustrate its use. After that, the HemView camera implementation is discussed in detail to show how it was worked into the application and the difficulties encountered. Once the architectural discussion is finished, the framework is analyzed for performance and general use. After knowing the performance limitations, future work ideas are discussed as possible extensions to the application. Finally, the last chapter reflects on the results and discusses the failures and success of the framework.

CHAPTER 2

EXISTING SOLUTIONS

There are many solutions available for working with cameras that are both generic and extensible so that they can support current and future camera systems. It is also important to point out that there have been previous attempts at creating a multi-channel aware camera application. Unfortunately, each of the solutions share the same flaw of only supporting single-channel cameras, but they have played a role in the direction of this architecture. Each of these systems are described in some detail below. It should be noted that many of these systems are open source so that it is possible to look at their capabilities and how they are put together.

2.1 Previous Implementation

There was a previous attempt at providing an interface for multi-camera systems. This solution was provided specifically for the HemView system [1]. The solution worked properly, but had many problems that introduced a need for a new kind of system. The two major problems were a lack of flexibility and disorganization within the code itself.

The first problem was a lack of flexibility. The system was specifically targeted for Point Grey hardware and did not consider the possibility of working with other camera types or vendors. There also was no support for other types of cameras. It was only concerned with functioning with the HemView camera and did not consider other types of systems. This reflected the second problem involving code organization.

The second problem was disorganization within the code. This hurt the maintainability of the system and also affected its flexibility. In terms of maintenance, it was very difficult to find out where things needed to be changed to introduce new features. In addition, finding and squashing bugs

was much more difficult. This lack of maintainable code also factored into why the flexibility was so poor. It resulted in a highly specific camera interface description, making it difficult to extend the capabilities of the system.

2.2 OpenCV

OpenCV is an image processing and computer vision library [2]. It is not really surveillance software, but is utilized by many solutions to support video processing and analytics. It also provides many examples and these are constrained to single image camera systems. This architecture builds on top of what OpenCV provides to add support for multi-channel camera systems.

2.3 ZoneMinder

ZoneMinder is an open source surveillance package that is highly configurable and supports many different camera systems [3]. It can also do post-processing of video that is on a local disk. It has the same basic constraint of supporting only single-image cameras. It does, however, provide an excellent example of bringing several different cameras to the table and helped to form some of the class structure in this system.

2.4 iSpy

iSpy is another open source surveillance product [4]. It provides a much more well-rounded set of features than products like ZoneMinder. It also has the same constraint of only working with single-image cameras.

2.5 Multitude

Multitude is like OpenCV in the sense that it is a library and not a full product [5]. It is also not specific to cameras, but supports many other items that are useful for multi-touch designs. Its class structure helped to define

many interfaces found in the current implementation of this system. It does, however, continue the trend of single-image camera support.

2.6 sentry360

sentry360 is a camera provider of 360 degree camera solutions using a single sensor [6]. They provide a software package to work with their camera system, but there is no evidence of support for multi-camera systems.

2.7 Point Grey

Point Grey is a camera solutions provider and creates many devices ranging from single images up to hemisphere displays using many camera sensors [7]. Point Grey's hemisphere offering is called the Ladybug [8]. The Ladybug comes as a full solution, however, and is not extensible beyond what Point Grey provides, from both a hardware and software perspective. In that respect, this shows a different limitation than the open source projects described above in that the solution is hard-coded for a specific device. In that way, it is much like the original HemView software offering.

CHAPTER 3

LIBRARIES

There are a few libraries that were leveraged to make the application possible. The most important is Qt, which provides the OS abstraction and GUI for the application. OpenCV is the other library and provides the data abstraction that is used to pass images around inside the framework. This chapter will explain how these third party libraries find themselves integrated with the application.

3.1 Qt

The application is built on top of a set of libraries called Qt [9]. Qt is a cross-platform GUI solution which supports operating systems such as Windows, Linux, and Mac OS X. It also provides a set of utility libraries to support development of an application. This library is heavily utilized by this project to both make it available on many platforms and to speed up development time.

In addition to cross-platform libraries, Qt provides a cross-platform build environment called "qmake". This uses native build tools to do the actual build, but allows the application developer to use an abstraction in order to support these different build environments.

The architecture of the application also follows that of Qt Creator [10]. Qt Creator is built around the idea of plug-ins. In much the same way, this application takes a similar approach in order to handle the various platform-dependent ways in which to talk to cameras. For example, on Windows, the Point Grey FlyCapture SDK is utilized to talk to cameras whereas on Linux, libdc1394 is used to accomplish the same goals [11, 12]. Qt Creator also served as a vital a reference implementation of both how to use the qmake build system and how to work with XML and plug-ins.

This section will continue to talk about the specific libraries that are utilized by the application to provide the final user interface. These include the Qt Core, GUI, and XML libraries.

3.1.1 Qt Core

The Qt Core library is made up of many utility classes utilized by the various other libraries. These classes also provide abstractions to items that vary by the compiler used or the operating system the application runs on. It provides standard objects such as vectors, maps, and threads along with synchronization objects like semaphores and mutexes. This is also where the translation capabilities live, not to mention the signals and slots mechanism that is the core of the framework.

This system uses this library extensively to support camera image capture. For example, a thread is spun off that handles image acquisition for each camera connected to the system. Locks are put in place to support synchronization of the various cameras in a multi-camera system. Vectors are used to buffer incoming and outgoing images. Altogether, a general system architecture was built using just the core library.

The application frontend uses the translation piece everywhere. Qt provides another tool called 'Qt Linguist' that supports translating a set of strings in the code from one language to another. This requires the use of a translation function throughout the code and the Qt Core library houses this functionality.

3.1.2 Qt GUI

Qt GUI is where all the windows and controls come from in Qt. The user interface pulls many things from this library to do things like display the video streams, do some OpenGL work for three-dimensional interfaces, and provide the menus and pop-up windows that support things like camera configuration. There is also some additional work involving image manipulation that this library provides, although much of this capability is taken from OpenCV.

The main object used for displaying single-image camera streams is the

'QLabel'. This class allows the application to display single images and when it is continually replaced, a video is shown. The interface is not restricted to just this kind of display, though. In order to support any kind of video stream, the system goes up a layer to 'QWidget' objects. This interface is also provided by this library and enables the system to support any kind of display a camera system may need.

3.1.3 Qt XML

The main purpose for the Qt XML library is to provide a mechanism for parsing and traversing the various XML configuration files used by the system. The main example in the core application is in the parsing of plug-in files and layout configuration files. External plug-ins, like the HemView plug-in, also make extensive use of XML to provide configuration information. Other options do exist for handling configuration, but the tree structure of XML proved to be the best option.

3.2 OpenCV

OpenCV is a cross-platform image processing and computer vision library that comes complete with interfaces for managing and manipulating images. This library is part of the core processing pipeline in the system that enables passage of data from one part of the system to the next. This enables a lot of flexibility in terms of what data is passed between each stage thanks to the matrix class in OpenCV. This also means that many of the stages can build on top of the capabilities in OpenCV. This allows for easy implementation and integration of algorithms written with OpenCV. OpenCV is also leveraged for its core image algorithms in the plug-ins for such things as debayering, image crop, and image masking.

CHAPTER 4

SOLUTION

The application utilizes a generalized approach to a camera. It looks at a camera as potentially having more than one output and allows these outputs to be interpreted in any way needed by the camera provider. This can range from anything simple, like an image, all the way to vector data needed for tracking algorithms. It also supplies an easy way to interpret the final pieces of information for purposes of display. This is all accomplished through the use of a plug-in system. For the purposes of this chapter, however, the focus will be on the core components and pieces that provide the flexibility for both the camera and display components.

4.1 Core Architecture

The core architecture for the application relies on five major components. The overall container is known as the filter tree. Its purpose is to maintain the flow of data and manage each of the components that make up the image pipeline. The camera abstraction is used to act as the source of the image pipeline to provide data for the rest of the tree to interpret. Filters sit between the source and sinks of the tree and serve to manipulate the image data as it passes through the pipeline. The data abstraction helps to define the interface by which each of the components operates on the data provided by the camera source. Finally, the display acts as the sink of the tree and provides a rendering surface for the final image data after going through all the filters. Each of the following sections goes into further detail about these five core architecture pieces.

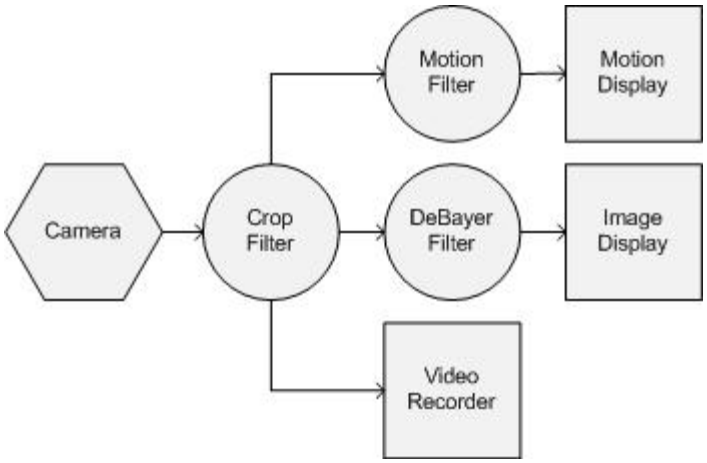


Figure 4.1: Filter Tree Overview

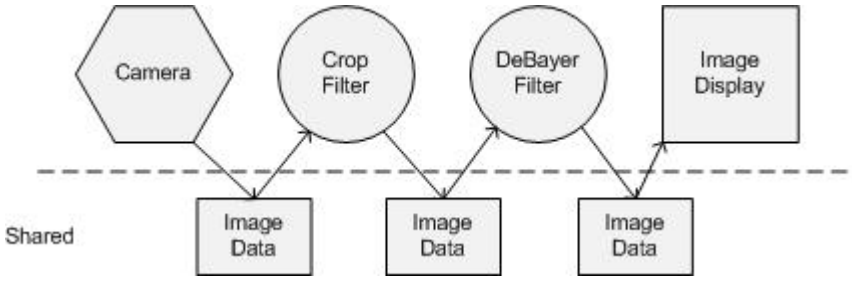


Figure 4.2: Component Data Sharing

4.1.1 Filter Trees

The application is built around the central idea of a tree. Figure 4.1 shows how cameras make up the root, displays and recorders are the leaves, and how each path from the root to a leaf may or may not have a filter. Each parent node passes its information to the child nodes via a shared data area. Each child shares the same data from the parent and uses it to either filter it and pass it down further or to act as an endpoint, such as a recording device or display. Figure 4.2 describes how the components of the tree share their data with each other. The data at each stage is abstracted such that it can represent anything as long as the parent and child agree to its meaning. In the case of Figure 4.2, each stage agrees the data is that of an image.

One of the key pieces of the architecture is how things make it through the image pipeline. To begin the discussion, we must first define the idea behind a 'tick'. A 'tick' represents a single execution cycle for a node. These ticks occur using a breadth first search through the tree starting at the root. The idea is that all nodes at a single level will have their tick before the nodes at

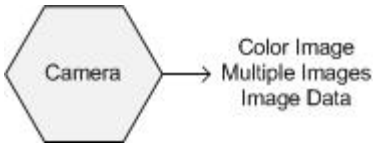


Figure 4.3: Camera Source

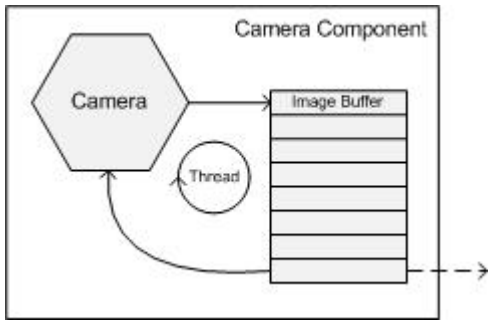


Figure 4.4: Camera Component

the next level execute. A parent produces data by placing it into the shared data area every tick. The child nodes will then analyze this data on their tick and either place it into their own output area or send it to a display. Once all leaves have had their chance, the process starts all over again for the next frame.

For the sake of simplicity, these trees only allow for a single parent. It is, however, possible to simulate the behavior of multiple parents and synchronize the production of these parents at a shared child node. This is done in the existing code base specifically for the HemView camera system.

4.1.2 Camera Abstraction

The camera abstraction provides a basic interface for all cameras to implement. In essence, the camera is a source node that provides some data, as described in Figure 4.3. Any camera plug-in must provide an implementation of this abstraction so that they can properly provide their output. Using this interface, many helper utilities were created to make it easier to introduce cameras into the system. For example, a camera component, described in Figure 4.4, was created for the filter trees so that images can be acquired in a separate thread from the rest of the filtering. Images are buffered until the component's tick, at which point they are moved into the shared space to be picked up by a child. All of this allows camera plug-in providers to be obliv-

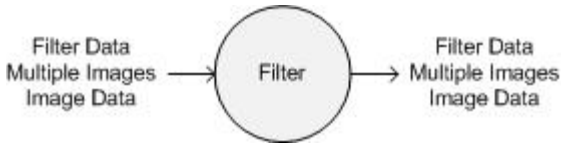


Figure 4.5: Filter

ious to the filter tree unless they have to provide custom functionality. For example, the HemView camera makes use of a series of cameras to produce the final image set. Thanks to the camera abstraction, this is all supported without any external modifications to the framework.

4.1.3 Filters

Each filter is required to take in an input and provide an output. Figure 4.5 shows the basic idea of what can go into and come out of a filter. In this process, it may do nothing or it may do some image manipulations such as cropping or debayering the image. In any case, it acts as a middle man between the camera and the final display inside of the filter tree. An important distinction for these filters, though, is that the provided image may not be a single-image. This makes the job a bit harder for filters since some cameras may be made up of several cameras to produce a single-image. The core library in the application provides two debayer filter implementations to show how a filter can be implemented to handle the multiple image case.

The two stock debayer filters take completely different approaches to solving the multiple image problem. The first debayer filter is designed to go through each image in series and convert it using OpenCV functions for color conversion. The second filter has the same capability, but it uses the QtConcurrent API to go through the images using multiple threads to take advantage of the CPU's multi-threading capabilities. Both filters work as expected and their performance is detailed in the analysis chapter.

As an example of extending the core library, the HemView plug-in provides a couple of filters to aid in rendering its own images. These filters include a crop filter and a mask filter. The crop filter utilizes OpenCV functions to get a sub-image from the provided input image. The mask filter uses information from a mask file to apply a mask to the input image. Both of these are capable of working on multiple images to make sure that the image

pipeline will work as expected.

4.1.4 Data Abstraction

The framework uses an abstract data representation based around OpenCV matrices. This allows each step of the filter tree to create and analyze the data as required. As an example, the matrices can represent both image data and general data that can be used for motion detection. The parent node defines what this data is and the child nodes must understand the data before hand in order to work with it. This section goes into more detail about the two common data types, image and filter data, used in the framework.

Image Data

The general, and default case, for the data abstraction is that the matrices represent image data. Knowing this, filters can be applied to do different image manipulations such as white balance or image cropping. The container for this data is responsible for defining what the image format is so that the proper transforms can be applied by downstream filters. As an example, the bayer pattern from a raw image is important to have so that a debayer filter can properly convert the image to its color form.

Filter Data

Another interpretation for the data is that it represents features or other data points that a filter can use to apply some computer vision algorithm. The common requirement is for motion detection. The infrastructure allows a filter to take in an image and produce the features as an output in the form of a matrix. This can then be given to another filter that uses this information to produce a decision and act on it. The end result is a flexible data transport system for many computer vision algorithms.

4.1.5 Displays

The main sink node for the filter trees are called displays. These displays can either render the final image or be used to do a recording of the data. Figure

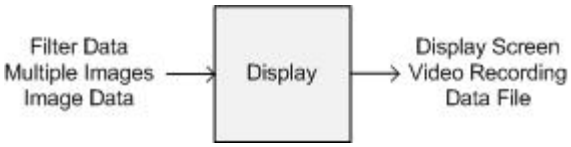


Figure 4.6: Display Sink

4.6 shows how the display sinks interact with the incoming data and some possible things they can do with that data. There are other ways these can be implemented, such as sending the images over the network to a waiting server or client that wants to see the live stream from a camera, or perhaps doing nothing with the data and only serving as a terminator.

Rendering

The basic case is that the display renders the image in a window. This is a broad concept and can include specialized displays for cameras that have one to many image sensors. This is an extremely nice concept as it allows for displays to provide 3D renderings or to do a simple, single-output display.

Recording

An important side use of displays is to do recording and saving of the image data to a file for playback later. It is left to the display to handle video encoding by whatever method the user desires. The main reason for this is that specialty cameras with more than one image will require special handling when being recorded to a file as the general encoder is not designed for this.

4.2 Layout Configuration

One of the extra features that came into play during development was an ability to configure the layout of the camera windows. As part of this task, it was decided that an XML configuration file provided the best way of defining the layout so that others could easily define their own layouts that could then be realized in the user interface. The definition is simple, but provides a lot of power to the end user.

```
<?xml version="1.0" encoding="UTF-8"?>
<displayLayout name="1">
    <display position="0" row="0" col="0"
        rowSpan="12" colSpan="12" />
</displayLayout>
```

Figure 4.7: Example Layout Configuration File

The best way to describe the layout description files is that the user is working with a grid. They can define the number of displays and the number of rows and columns each of those displays take. The end layout is then taken into the main part of the user interface and is given a large area to expand into. Each row and column is given the exact same width and height, so the number of each decides the final width and height of the display. Knowing this, it is possible to write a layout XML file for a specific use case.

The layout XML file has very few elements that are required. Figure 4.7 shows a basic single display configuration. This is a very simple layout, but it is not much more work to define display layouts such as a single window surrounded on two sides by five displays, or perhaps two large displays next to each other with two rows of four displays below. In any case, the layout works the same, allowing the user to drag and drop cameras around to make whatever arrangement they wish.

4.3 Plug-in Architecture

Much of the architecture is based on the idea of plug-ins. The reason for this is to allow the higher level core architecture to ignore the specifics of each camera and how to display them. No cameras are provided by default, but some displays are provided, such as a single-output display. For more specialty cameras, such as the HemView camera, custom displays can be provided to properly display and manipulate a camera view. At the same time as providing this flexibility, the plug-in system allows vendors to create their own, private plug-ins that do not require the vendors to expose their source code.

```
<?xml version="1.0" encoding="UTF-8">
<plugin name="plugin_name" version="plugin_version">
</plugin>
```

Figure 4.8: Basic Plug-in Configuration File

4.3.1 Plug-in Discovery and Loading

When the application starts up, it scans a plug-in directory to find all available plug-ins. The plug-in itself is made up of two files: an XML file that describes it and the actual shared library that provides the implementation. In order for the plug-in to be discovered, the XML must be well formed and contain at least the following information.

There are many additional fields that can be provided that are more useful for informational purposes. These include vendor, description, and a URL for the plug-in. The parser also understands dependencies, but these are unused at this time.

If the XML file is properly loaded, the plug-in itself will be loaded from the accompanying shared library. The library must be inside the plug-in directory and its name must match that of the 'plug-in_name' field shown in the XML example in Figure 4.8. The plug-in is then loaded using the Qt mechanism to load a plug-in from the shared library. The new plug-in must extend the 'IPlugin' interface provided by the core library. If it does not extend the proper class, it will fail to load and an error will be propagated up.

Once a valid plug-in has been loaded from the library, it is set to the side for later initialization. Once every XML file has been visited, the system continues on and calls the initialization function for each plug-in. It is at this time that a plug-in registers its capabilities with the rest of the system.

4.3.2 Plug-in Registration

When the plug-in is initialized, it is allowed to do whatever it needs to do to get itself ready to go. This can be used to do some early initialization for the system or to just get itself registered with the core plug-in system. This is an important time because this is where the plug-in places itself alongside the other plug-ins so that the core system can make use of them. In some cases,

this process can make a plug-in available to other plug-ins in the system.

The special case of a plug-in needing to use other plug-ins does exist and is leveraged, for example, by the HemView camera plug-in. In that case, it is possible for a plug-in to give itself a priority. This is useful for camera plug-ins that need to look for other cameras, as is the case for HemView. The priority makes it so that cameras with a lower priority are searched for before the higher priority ones. This allows the higher priority cameras to use the list of lower priority cameras when doing their own discovery.

4.3.3 Plug-in Types

There are two types of plug-ins recognized in the system. The first consists of camera plug-ins that provide listing and access capabilities for a certain type of camera. The second is a display plug-in that provides a display for some class of camera using a mimetype mechanism. Each type of plug-in registers itself in a different way and provides different capabilities to the end system.

Camera Plug-ins

Camera plug-ins are pure source nodes in the filter tree architecture. Their role is to grab an image from a hardware device, convert it into the OpenCV matrix space, and then return this data to the rest of the system. There are two pieces required to accomplish this goal and they include a camera factory and a camera instance.

The camera factory is what is used to register the camera plug-in with the plug-in architecture. In order to register, the plug-in initialization is expected to add this factory to the camera manager list taken from the core application. The factory itself is then responsible for searching for available cameras, providing this information to the core application, and also creating camera instances to interact with the physical cameras. The process of camera discovery varies from vendor to vendor, so this is left to the plug-in. In some cases, this will involve looking at configuration files and in others, it is just a simple library call to scan for hardware devices. Once all known cameras are listed, the factory is called upon to create a camera instance that is capable of grabbing images.

The second part of camera plug-ins is to have an implementation for interacting with the physical cameras. The most basic of capabilities is to grab an image from the camera and return it as an OpenCV matrix. This matrix is used by the rest of the system and eventually finds its way to a display. What needs to be mentioned is that this part of the system is somewhat flexible. By returning an image, it really means that it can return a set of images. This is important for camera systems that make use of more than one camera to produce an image, but still allows for regular, single-image cameras. Besides that, this instance can also be used to modify camera attributes such as shutter speed or color balance and potentially adjust pan, tilt, and zoom parameters.

Display Plug-in

Display plug-ins are the opposite of camera plug-ins. They act as the sink nodes in the network and provide a way for displaying an image or a set of images, depending on what kind of camera they are for. Just like camera plug-ins, display plug-ins are made up of two pieces: the factory that creates them and the displays themselves that take care of doing the actual display work.

The display factory is used to register the plug-in, determine if a display supports a particular camera, and to create display instances to use in the application. A separate display manager is used when registering a display plug-in and the plug-in is expected to add the display factory to the manager's list when it is initialized. When determining if a certain camera is supported, the display manager uses a mimetype mechanism from the camera to determine if a display is available. The display factory provides a function that checks if it has a display that supports that camera type. If it returns true, the factory must be able to provide a display instance that is capable of handling images from the camera.

The display instance itself is responsible for properly interpreting the camera image data and displaying it in a widget suitable for a Qt GUI. The display is expected to manage a Qt widget object that it uses to update with images from the filter tree connected to it. The only other function provided by a display is a way to take a snapshot of the current image being shown in the widget. Besides that, the inner workings of a display plug-in are left to

the implementation.

CHAPTER 5

EXAMPLE PLUG-INS

Several plug-ins were developed for the application that utilized the aforementioned plug-in infrastructure. These included a plug-in for cameras from PointGrey, another for IP cameras, and a third for the HemView camera. This chapter seeks to explore some of these plug-ins in an effort to show how the infrastructure can be leveraged to create vastly different experiences depending on what cameras are connected to the host system. The HemView camera has a dedicated chapter, so this chapter focuses on the simpler plug-ins.

5.1 General Plug-ins

The application itself uses the plug-in architecture to add some basic functionality that can be expected of these kinds of systems. The main purpose of this was to provide a general camera display that could show a single camera image stream in the user interface. Many plug-ins, like the IP camera and PointGrey plug-in, can leverage this display to show their image streams. The other purpose of this was to show how common functionality can be added to the main application and be provided to plug-ins that need it.

5.2 PointGrey Cameras

PointGrey offers an SDK for communicating with both their network cameras and their firewire cameras. In order to make use of these cameras, a library was needed to provide the translation layer between the SDK and the general Qt application. Thanks to the general plug-ins provided by the core application, the only thing required from this plug-in was a way to get access to the cameras.

For the purposes of the application, this plug-in focused squarely on firewire cameras. Access to the network cameras requires a slightly different interface that was not necessary for the project, so only firewire support is currently available. To accomplish this, the PointGrey plug-in provides a basic camera factory that uses the SDK to provide information about connected cameras. The SDK is very helpful in accessing this information, so the only hard work was to convert the PointGrey SDK information into a camera instance usable by the video framework.

PointGrey also makes it straightforward to get images from connected cameras. Since this is the main purpose of a camera plug-in, it was simple to do and only required translating from the SDK specific images to the required OpenCV matrices. There was, however, one difficult portion that had to be dealt with and this had to do with the PointGrey firewire camera driver.

During development, it was determined that the driver is not thread safe and appears to like to reuse buffers at the kernel level when grabbing images. Accessing many different cameras as often as the application needs to eventually caused blue screens. The solution to this problem is to use some locking mechanism. A normal mutex locak was not adequate to protect the driver. Evidence of this was found out as only certain camera pipelines would continue and some would block indefinitely. The solution required a mix of a mutex lock and a FIFO buffer of semaphores so that each image request would go through, but the actual driver access would be guaranteed to be serial.

5.3 IP Cameras

In an attempt to prove the flexibility of the system, a second type of camera was needed. Since the major scope of the project involved security systems, network cameras, also known as IP cameras, were selected as a secondary camera. The selected cameras have a standard interface for talking to them and this information was used to create the IP camera plug-in. The camera interface can be found by contacting TRENDnet [13].

While implementing the camera plug-in for IP cameras, it was determined that scanning for these cameras was not a viable option. Each camera also

```
<?xml version="1.0" encoding="UTF-8"?>
<ipcamlist>
    <ipcam name="Doorway" ip="192.168.1.56" port="80"
        username="admin" password="admin" />
</ipcamlist>
```

Figure 5.1: IP Camera Configuration File

required authentication information. This forced the plug-in to rely on an XML file that described the cameras available on the network along with their authentication information. Camera specific parameters, such as width and height, were pulled directly from the cameras, so the configuration file was kept short and simple. Figure 5.1 shows an example configuration file for IP cameras.

The cameras use a basic HTTP interface for communicating and sending information to different clients. Qt provides many mechanisms for talking with HTTP servers and these were used to grab the image data from the IP cameras. In fact, the response data was able to be directly fed into the Qt image APIs to create the final image that was sent to the application display window. Together with the PointGrey cameras, this plug-in helped to show some of the benefits of the plug-in architecture. This was taken even further as special purpose cameras came into the picture.

CHAPTER 6

HEMVIEW CAMERA EXTENSION

The HemView camera was a major component during development of the application and had a much greater impact on the design and implementation than any of the other cameras. This chapter reflects this and provides greater insight into the insides of the HemView camera extension.

6.1 HemView Camera

The entire application architecture was aimed at enabling special purpose cameras. Not only did the system aim to make it easier to gather images, but it also made special purpose displays more achievable. The key component used to frame this architecture was the HemView camera. The HemView system is made up of seven separate cameras provided by PointGrey and produce a final image that requires a 3D rendering to view the final result. The flexibility of the image pipeline enabled this camera to function properly and provide final display to the user.

The first step in implementing the plug-in was to find some way of describing which cameras make up the array. To accomplish this, an XML file format was created that specified each of the cameras in the system such that each one could be found using another plug-in that is capable of finding single cameras. In the primary case, the camera provider was PointGrey, so each ID in the configuration file uniquely identified each camera that made up the array. Now, to enable finding these cameras, this plug-in needed to load later than all of the other single camera plug-ins.

Taking advantage of the plug-in loading priority provided by the plug-in framework, the HemView plug-in made it so that it loaded after all other plug-ins. This allows manufacturers to design the camera with any set of sensors in the array. Since the configuration file uses a general ID for each

sensor and the HemView plug-in loads last, the architecture can pick up on any sensor combination to setup the HemView system. In addition, the general camera interface allows the plug-in to be generic when setting up the array and capturing its images.

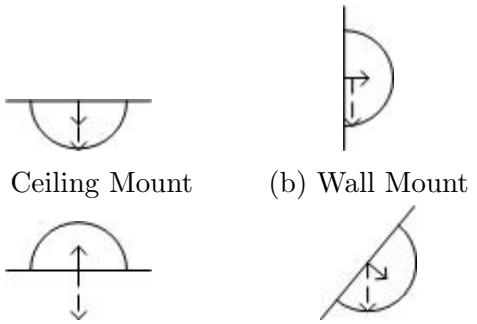
6.1.1 Image Capture

HemView cameras provided an interesting issue when it came to grabbing a single set of images using all of the cameras in the array. The framework's video pipeline was designed to easily support cameras that use more than one camera, but it did not provide a mechanism for synchronizing the output of multiple cameras to create a single image set. To accomplish this task, the HemView plug-in provided some classes to support synchronizing a set of images. The idea behind them was to take a single image from each camera and call that a synchronization point. When this happened, a single image was provided to the pipeline to send down for further analysis by any attached filters. This locking behavior was accomplished with a semaphore so a full image set was acquired as soon as each camera in the array provided an image. Mutexes are used to protect against the case of one camera providing images faster than another. This all helped to ensure that only one image was taken from each camera whenever a set was created. The only thing left to do with an image set was to now display it.

6.1.2 Image Display

HemView cameras have an interesting requirement to display all the images from the array as a hemispherical image in a 3D space. In addition, the user needed to be able to manipulate the image and explore it as if they were working with a PTZ camera. Thanks to the plug-in infrastructure, custom camera displays are enabled and so the HemView plug-in provided its own way to display the image set. The result is an OpenGL display that takes advantage of Qt's capabilities to work in 3D. The display enables the user to pan, tilt, and zoom through the 3D rendering and provide the proper stop points and experience that the user expects.

An important note to make about the display plug-in for the HemView



(a) Ceiling Mount (b) Wall Mount

(c) Table Mount (d) Angle Mount

Figure 6.1: Different Camera Mounting Options

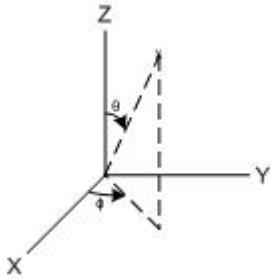


Figure 6.2: Spherical Coordinate System with Radial Distance of 1

camera is that the user interaction with the display depended on how the camera itself was mounted. For instance, a camera put in the ceiling would need to maneuver differently than one that was placed on a table or mounted to a wall. Figure 6.1 shows the different camera mounting positions considered by the display plug-in. The longer, dashed arrow in the diagram shows the gravity vector, the solid arrow represents the initial view, and the hemisphere shows the viewing area of the final image. To support the different mounting configurations, the HemView configuration file can specify the mounting angle so that the display can account for the way the camera was mounted. Knowing the mounting angle only solves half of the display problem, however, and the second half is determining when to stop the users panning and tilting.

One way to imagine interacting with the HemView camera is that is acts much like a PTZ camera. This means the user should not be able to turn the camera upside down when looking at an image. This posed an interesting problem when dealing with things in the OpenGL domain. To begin, consider a spherical coordinate system with a constant radial distance of one as shown in Figure 6.2. This system has to be rotated and interpreted a little differently

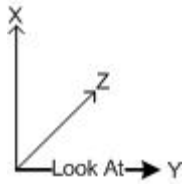


Figure 6.3: OpenGL Look-at Coordinate System

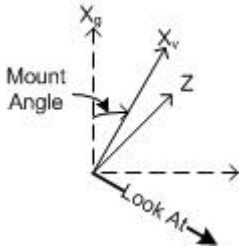


Figure 6.4: Gravity Versus the Virtual Coordinate System

in order for the HemView meshes to work cleanly in OpenGL.

The meshes for each HemView camera are generated with a certain expectation of the look-at vector in OpenGL. The look-at vector defines the initial OpenGL space which includes which way is up and where the initial view is located. This new space is shown in Figure 6.3. As Figure 6.3 shows, the up vector is the x-axis and the look-at vector is the positive y-axis. This leaves the positive z-axis going into the page. This defines how things will operate in the virtual world of the display, but does not fully define how the physical camera is expected to operate.

The other item to consider is the physical coordinate system for the camera. In the physical world, the camera is expected to rotate about the gravity vector. We will continue to reference the gravity vector as if it is the x-axis with the horizontal plane being y and z. For the sake of simplicity, it is assumed that the z-axis in the virtual world is coincident with the z-axis in the physical world. This equates to there being no yaw when a camera is being mounted. Pitch and roll, also known as the mount angle and mount rotation respectively, are handled by the display through the camera configuration file. Figure 6.4 illustrates this new coordinate system and how it differs from the look-at coordinate system from Figure 6.3. The dashed axes represent the physical coordinates and the solid axes show the virtual. Going back to Figure 6.1, the gravity arrow shows how the rotation axis varies with the mount angle.

$$x = \sin(\tau) \tag{6.1}$$

$$y = \cos(\tau) \tag{6.2}$$

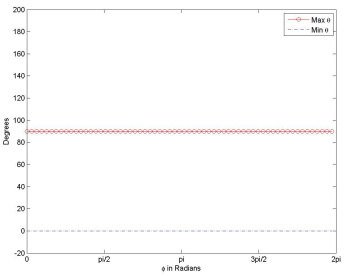
$$z = 0 \tag{6.3}$$

With all the axis information sorted out, it is a matter of trigonometry to define what the rotation axis should be in the virtual world. Since we make the assumption of no yaw, the gravity vector must lie in the XY plane in OpenGL. The actual rotation axis is computed using the equations shown in Equations 6.1 through 6.3 where τ represents the mount angle. The sin and cos are swapped to x and y respectively due to the rotation of the look-at vector in OpenGL. Because of the lack of yaw, z is able to be a constant of zero.

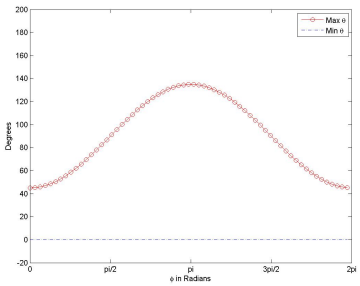
Defining the stop points involves a bit more trigonometry and some tricks, but the calculations are straightforward. The general idea is that the interface should stop the user from turning the camera upside down in the OpenGL domain. The display must also keep the user from looking outside of the camera's dome. This means that both θ and ϕ must be limited.

The θ limit is controlled by the mount angle and ϕ. The calculation turns out to be proportional to the mounting angle and the cosine of ϕ. Intuitively, the amount of allowed rotation with respect to the gravity vector ranges from zero degrees all the way to 90 plus the mount angle in the biggest case ($\phi = 180$) and zero to the mount angle in the smallest case ($\phi = 90$). Everything in between those two values of ϕ will vary based on a cosine function. To illustrate this point, Figure 6.5 shows how the θ min and max vary for different values of ϕ and the mount angle.

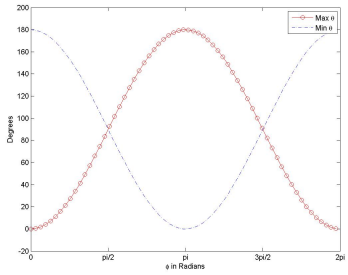
An interesting case to talk about is that of the mounting angle of 90°. In this case, the minimum and maximum values of θ cross paths, as Figure 6.5c illustrates. The window where the min and max values are valid, the values between $\frac{\pi}{2}$ and $\frac{3\pi}{2}$, specifies the valid range for ϕ in the OpenGL space. In all other mounting angles, ϕ is able to vary from zero to 360. What should be noted is that the min and max θ values do help to define how much ϕ can vary. For instance, in the 45° mounting case, if θ is currently 100°, ϕ cannot change such that 100° becomes an invalid value for θ. This puts a limit on ϕ and keeps the user from rotating out of the hemisphere.



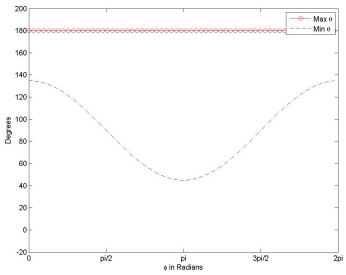
(a) Mount Angle = 0°



(b) Mount Angle = 45°



(c) Mount Angle = 90°



(d) Mount Angle = 135°

Figure 6.5: θ Min and Max with Different Mount Angles

The result of all this work is a much smoother and intuitive experience for the user. The user cannot rotate out of the display and the rotation properly aligns with the user's expectation of gravity. The end result is an experience similar to the user poking their head out of the ground and looking around. The next key component of the display was trying to manage the amount of resources the OpenGL rendering required.

In order to render the images in OpenGL, several textures were created and applied to a series of meshes. Since this rendering happens quite often, it is important to minimize the amount of textures being created and destroyed whenever an image is shown. The display takes advantage of texture replacing functions provided by OpenGL and keeps the number of created textures equal to the number of cameras in the array. When it came to rendering the actual meshes, a display list was used to make it easier and faster to repeatedly draw the scene. To further limit resource usage, the OpenGL display had to take into account how it would be used by the general application.

In the design of the user interface for the software, it was required that each camera could be duplicated to another display in the same layout. This posed a challenge for the HemView display as each display would require more

textures to be created for each new display in the interface. This problem was solved by simply making each new display share the same texture context as the original window. Qt was leveraged to make this happen thanks to its support for resource sharing between OpenGL contexts. A nice side effect is that each of the meshes could now also be shared among the OpenGL contexts. This all helped to minimize how many resources were utilized whenever a HemView camera was displayed.

CHAPTER 7

ANALYSIS

The final deliverable of the project is an application that is flexible and performance oriented. The purpose of this chapter is to put these concepts into numbers and examples to illustrate how well the system performs under common usage models.

7.1 Testing Setup

A basic development machine was used to produce the performance numbers found in this chapter. This section outlines the various software libraries and hardware components that made up this system. In addition, the camera used to capture the live images is described in more detail.

7.1.1 Libraries

The following release libraries were used to compile the software for performance analysis.

- Qt 4.7.2
- OpenCV 2.2.0
- FlyCapture 2.1.3.5
- Windows SDK for Windows 7 (7.1.7600.0.30514)

Due to limitations in Qt for Windows, all created binaries are for a 32-bit machine.

7.1.2 Host Machine

The host machine was a Windows 7 Professional SP1 64-bit system with all recent updates as of writing. The CPU was an Intel Core i7 920 with a clock speed of 2.67 GHz. It is a quad-core CPU with hyper-threading enabled. The system had a total of 6 GB of RAM installed. The GPU was an NVIDIA GeForce 9500 GT with a GPU clock of 550 MHz, a memory clock of 702 MHz, and 512 MB of graphics memory.

7.1.3 Cameras

The camera used to drive the system was a HemView system consisting of seven Point Grey Research Dragonfly firewire cameras. Each Dragonfly had a resolution of 1024x768 and was set to output 8-bit raw data for each pixel using a standard Bayer pattern. With this setting, each camera can output a maximum of 15 fps. The cameras were split into a group of three and a group of four and each group was placed on a separate bus with the use of firewire hubs. Each firewire bus was capable of handling 400 Mbps of traffic (IEEE-1394b).

It is important to note that the firewire bus does not provide enough bandwidth for all cameras to operate at 15 fps. This is a little surprising considering the math does not align with the results. For example, the amount of bandwidth used for one camera is computed as

$$\frac{1024 \times 768 \text{ pixels}}{\text{frame}} \times \frac{8 \text{ bits}}{\text{pixel}} \times \frac{15 \text{ frames}}{\text{second}} = 94.4 \text{ Mbps}$$

This means that each IEEE-1394b bus should be able to handle up to four cameras running at 15 FPS. This was not the case and can only be attributed to the overhead of communicating with the Point Grey cameras and the firewire bus protocol. The end result is that the cameras must be run one level below maximum at 7.5 FPS for the duration of the performance metrics. This means each camera takes half as much bandwidth, or 47.2 Mbps.

7.1.4 Measurement Tools

The main tool used for finding performance numbers was an additional library added to the code base. This provides a class that is capable of finding rates and timing how long operations take. The additional overhead of doing this is not enough to greatly affect the results of the measurements. In order to get CPU and memory utilization numbers, Windows Performance Monitor is used. GPU utilization is measured using GPU-Z [14].

7.2 Performance

To ascertain the performance of the system, there are various pieces that need their performance measured. This section goes into detail about the different core components and gives an idea of how well they performed. It also takes a look at more general measurements to define the minimum system requirements for the application.

7.2.1 Debayer Algorithm

The debayer algorithm is the core component when working with the HemView camera system. It is important that this piece performs well to make sure the maximum frame rate can be achieved. The primary measurement was a timer that counted the elapsed time before and after the operation. The debayer algorithm comes from OpenCV and so this measurement only serves to compare the two techniques available for the filter chain.

The first implementation was a serial version that debayered each image one right after another. Using the Qt time objects, the time it took to debayer seven images from the HemView camera was between 0.027 and 0.030 seconds per frame. In the worst case, this means the filter can handle 33.3 frames per second. This is more than adequate for the current HemView system which is running at 7.5 FPS and shows that it will work for the target of 30 FPS. This didn't take advantage of a multi-core system, so another version was produced.

In the second implementation, the Qt Concurrent framework was used to produce a parallel implementation so that each image in a set could be de-

bayered in parallel. Using this new method, frame debayer times were cut to 0.09 to 0.012 seconds per frame. This shows a markedly improved performance to 83.3 frames per second. Keep in mind that this is 83.3 HemView frames which each contain seven images. This gives the filter a much better buffer as the images get more and more complex. For that reason, this filter was chosen and remains the default debayer filter.

7.2.2 Filter Tree Timing

The next important piece is how long it takes an image to be taken from the camera and provided to the display to be shown. This is handled by the filter tree mechanism. For each firewire camera, the trees took anywhere between 121 and 140 ms. Each tree has a camera component followed by a crop and a mask. The final stage is to put it all in a join component that synchronizes the camera images for the HemView system. Interestingly, most of the time is spent waiting for an image from the camera. In most cases, this takes over 90% of the time. The next closest component only takes about 3 to 4 ms to execute. This result is not unexpected seeing as the camera is only taking images at 7.5 FPS, or one image every 133.3 ms. This does mean, however, that the camera component's buffer effectively stays empty.

Going up a level to the HemView system's tree, it shows a very similar timing. Each image set is pushed through the tree in 125 to 140 ms. Most of the time is spent waiting for images as before, but more time is spent in the debayer filter and display than the other trees spent in cropping and masking. On average, the camera time takes about 70% of the time with the display taking the next largest piece at 18%.

7.2.3 Resource Usage

As a baseline to compare against the running application, the different memory, CPU, and GPU utilization numbers were recorded while running Windows with Aero effects enabled. Inside of GPU-Z, average was selected for each measurement. Three different modes were used to elaborate on how the architecture performs with the different viewing modes of the application. Table 7.1 shows how the application performed from the perspective of the

Table 7.1: GPU Performance Numbers

Measurement	Windows	One View	Two Views	9 Views
Temperature	53.0 °C	54.2 °C	54.0 °C	54.0 °C
Memory Used	186 MB	254 MB	249 MB	259 MB
GPU Load	1%	6%	5%	8%
Memory Load	4%	7%	7%	9%

Table 7.2: CPU Performance Numbers

Measurement	Windows	One View	Two Views	9 Views
Memory Used	2.335 GB	2.636 GB	2.666 GB	2.69 1GB
CPU Load	0.785%	23.5%	24.0%	24.5%

GPU.

As the results show, even as more displays are added, the resource usage increases slightly. This reflects the special care taken to minimize the affect of additional displays on the graphics hardware.

In a similar way, the memory and CPU usage data were collected using the average number provided by the Windows Performance Monitor. Again, each mode is compared against a baseline of just Windows 7 running. Table 7.2 outlines the results of running the application in various modes.

Once again, minimal increases in resource usage are seen even as nine views are added to the display.

7.3 Flexibility

Flexibility is a very subjective area when it comes to code analysis. This section attempts to discuss some of the measures made available by the system and goes into greater detail about how the HemView camera takes advantage of these abilities to make a very flexible platform.

7.3.1 Core Architecture

The core architecture is based on a plug-and-play system that allows cameras, filters, and displays to define their own interfaces. Nothing in the core pipeline limits these items. Cameras are free to define their interface, the

number of images they provide, and the data they output. For example, a camera can provide two channels of data such as color and depth, as is the case for depth cameras, or it can provide 7 images, as is the case for the HemView camera. Of course, the classic case of a single image is still possible.

When it comes to how the cameras connect to the computer, this is also left to the camera to decide. The interface may be firewire, USB, or Ethernet; they will all work provided the camera provider implements the proper interfaces. This makes it very simple to provide expandability for new interfaces such as Thunderbolt or any other interface technology released in the future.

When it comes to displays, the architecture only requires that the display outputs to some kind of QWidget so that it can be placed into the final layout. This enables displays to be very robust, taking in many images or one, using image data along with analysis information, and perhaps other ideas yet to be discovered. Filters are also flexible in the same sort of way. They can take in image data and output image data, or they can vary that information to transport computer vision data. The only real requirement for filters, and cameras for that matter, is that there is some node that can accept its output data.

7.3.2 Plug-in Infrastructure

The plug-in infrastructure is a bit constrained. In order to add new plug-in interfaces, it takes new classes that manage the new interface and must be provided in headers to enable plug-ins to tie themselves into the system. This can make it difficult to introduce new interfaces to the core application.

As a sort of comparison, one can look at the plug-in infrastructure used by such applications as Qt Creator. In their implementation, they allow plug-ins to toss objects into a general pool of objects. From there, plug-in interfaces can request objects of a specific type and can pull them from the object pool. This makes it very simple to extend the number of interfaces provided by the application. At the same time, more than one extension point can take advantage of plug-ins without the plug-in developer even being aware. This kind of flexibility was the goal, but it was not met due to time constraints.

7.3.3 HemView Camera

The main purpose of the software architecture was to support various camera configurations when it came to the HemView camera system. One of the first goals was to enable multiple HemView systems with just the use of different configuration files. The configuration file for the HemView system used for analysis is shown in Figure 7.1.

Figure 7.1 shows how the configuration file defines the ID for each camera in the system, the mesh file used when displaying the image, the region of interest in the camera's image, and the file used to mask the pulled images. The end result is shown in Figure 7.2 as the display on the right. The left camera shows the IP camera live feed.

The last camera described in Figure 7.1 represents the center camera in the HemView system. If that section is updated to reference the IP camera, the center camera can be switched. For example, that last section can be updated to look like Figure 7.3.

The end result of doing this is shown in Figure 7.4. The resulting image does not look quite right, but that is because the mesh and mask for the center camera were not designed for the IP camera. It does, however, show that the architecture is flexible enough to work with almost any camera configuration and still produce a proper image result.

7.4 Ease of Use

Ease of use is another subjective area to look into. This section looks a little bit into how the user interface enables users to work with the cameras and dives a bit into plug-in development.

7.4.1 End Users

The end user is expected to configure what cameras are available to them, fire up the application, select which camera to look at, and to interact with the cameras using the provided interface. All of these pieces play a part in the user experience and this section looks at these areas to analyze the ease of use for the user.

```
<?xml version="1.0" encoding="UTF-8"?>
<hemcam name="HEM08">
    <camera serial="7200233" position="0">
        <mesh file="7200233.bmo" />
        <roi left="108" top="26" width="816" height="738" />
        <mask file="7200233.raw" raw="true" />
    </camera>
    <camera serial="7200226" position="1">
        <mesh file="7200226.bmo" />
        <roi left="96" top="16" width="824" height="752" />
        <mask file="7200226.raw" raw="true" />
    </camera>
    <camera serial="7200228" position="2">
        <mesh file="7200228.bmo" />
        <roi left="100" top="22" width="816" height="740" />
        <mask file="7200228.raw" raw="true" />
    </camera>
    <camera serial="7200229" position="3">
        <mesh file="7200229.bmo" />
        <roi left="96" top="16" width="824" height="752" />
        <mask file="7200229.raw" raw="true" />
    </camera>
    <camera serial="7200230" position="4">
        <mesh file="7200230.bmo" />
        <roi left="100" top="16" width="824" height="746" />
        <mask file="7200230.raw" raw="true" />
    </camera>
    <camera serial="7200232" position="5">
        <mesh file="7200232.bmo" />
        <roi left="100" top="28" width="816" height="736" />
        <mask file="7200232.raw" raw="true" />
    </camera>
    <camera serial="7200227" position="6" master="true">
        <mesh file="7200227.bmo" />
        <roi left="96" top="14" width="816" height="730" />
        <mask file="7200227.raw" raw="true" />
    </camera>
</hemcam>
```

Figure 7.1: Example Configuration File

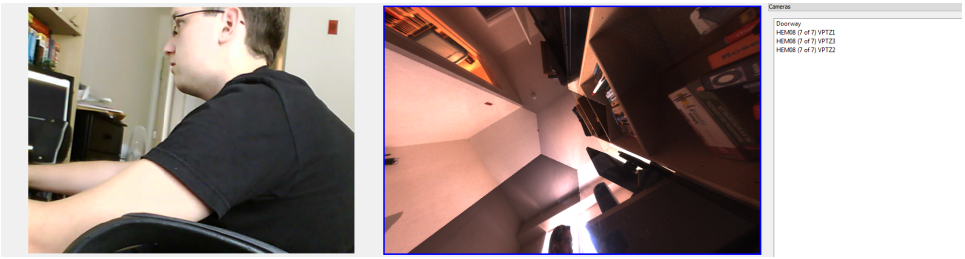


Figure 7.2: IP Camera along with HemView Camera

```
<camera serial="192.168.1.56" position="6" master="true">
    <mesh file="7200227.bmo" />
    <roi left="0" top="0" width="640" height="480" />
    <mask file="7200227.raw" raw="true" />
</camera>
```

Figure 7.3: Configuration Update for IP Camera

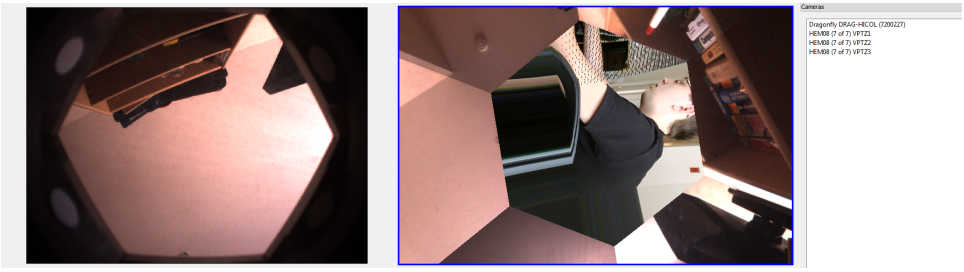


Figure 7.4: IP Camera Used as Center Camera of HemView

The provided base plug-ins in the architecture tend to use XML files to describe what cameras are available to the user. Many times, these files are user specific and require the user to fill them out so that the plug-ins can find the cameras. This process is a little intensive for users because of the XML configuration format. Each plug-in may have its own format as well and this can make the user's experience more difficult. In the end, it is not the best interface for ease of use and could use some analysis to come up with a more enjoyable user experience.

Firing up the application is very straightforward and works as any user would expect. All it takes is for the user to run the executable and the application pops up with a list of available cameras. This list makes it simple for the user to select which camera to display since it uses the names provided in the configuration files. When the user creates a configuration file, they are expected to use descriptive names and this helps to enhance the camera selection options. This is a bit out of the way, so it is not the easiest method for the user since they have to understand the flow. Assuming the configuration is done properly, selecting the right camera is very straightforward and only requires a single click.

The camera interaction controls are a bit interesting. They require shift and drag to copy a display or a control and drag to do a move. Once the user understands this control, it is very easy to use to manipulate how the displays show up in the final layout. There is also a drag image provided to help notify the user that a drag action is happening and a highlight shows where they are dropping the display. From an ease of use perspective, it is very easy to control with a mouse and keyboard. It might prove more of an issue if the camera control system does not provide this interface.

7.4.2 Plug-in Developers

The second half of the end user experience is to look at what the common plug-in developer will encounter when working with the plug-in system. As it stands, the interface is a bit convoluted. It requires several classes to get the plug-in registered and provide the classes to support them. The first required class is a plug-in loader that is required to register all of the plug-in extensions with their respective extension points. The next class is a factory

that creates an instance of the plug-in extension, such as a display or camera. The final piece is the display or camera itself. All of this takes a bit to get set up and even though most of the code is boilerplate, it can be a bit of trouble for the developer to get things up and running. Once this is taken care of, though, the developer is free to utilize the flexible nature of the architecture to produce whatever camera or display they need to.

7.5 Image Synchronization

One of the more important pieces relates to the HemView camera specifically. This section talks about how well the system maintains image synchronization. As it turns out, the current implementation shows a lag of approximately 1 to 2 frame between the two firewire buses involved. This is very noticeable to the end user. For example, moving a hand from one side of the camera to the other shows an obvious discontinuity as the hand takes 1 to 2 frames to show up in an adjacent image panel. This means that the images are not being properly synchronized between all of the cameras. However, each bus does show proper image synchronization and for that bus, transitions between image panels line up properly.

CHAPTER 8

FUTURE WORK

This chapter explores some of the possible extension work that can be done to make the framework better. Many of these items address common video application capabilities, but the difficulty of implementing these items is made greater because of the modularity of this framework's architecture. This does not mean they cannot be done, however, just that the time constraints kept these items from getting completed.

8.1 Camera Parameters

In general, many surveillance applications, and camera utilities in general, allow users to make modifications to camera parameters such as white balance or frame rate. The difficulty for this architecture is that whenever a feature is implemented, it must be based on the modular concepts used in the rest of the framework. To provide this functionality in the framework, a two-step approach is required and this section addresses these steps.

The first step is to outline a common set of camera parameters that are generally supported by camera vendors. Once these features are defined, the camera interfaces need to be updated so that the GUI can query the cameras for their capabilities using this standard set of feature values. Alongside of this will need to be functions for getting and setting these different values. The basic approach would be to use an enumeration of values and a single getter and a single setter. Once this is done, the next step can be addressed so that specialty cameras can give access to their special features.

This second step is a bit trickier as it requires another interface function for the cameras that outlines the extra features the camera provides. This comes in handy for cameras like the HemView system because it may be nice to define what color conversion it uses, or maybe the mounting angle needs to

be modified. The general idea is that the camera can let the GUI know how many extra features it has. From there, the GUI can query the camera for each parameter and build a GUI around the options the camera advertises.

8.2 Camera Support

The application currently supports only three different cameras. It is important for this kind of application to support as many different camera types and vendors as possible to increase the likelihood of its adoption. The existing plug-ins help to outline how to add these new cameras and show that the process is not too difficult.

What needs to be mentioned is that the implementation for camera plug-ins is a bit convoluted. There are several classes that have to be implemented in order to provide the proper support interfaces. It would be good to revisit this plug-in interface and look for optimizations to make adaptation simpler.

8.3 Layout Memory

One of the major missing features for the layout is to allow the application to remember which cameras were in which displays in the layout. The hardest part about adding this feature is creating a way to remember each camera so that it can be brought up as easily as possible when the application is restarted. One option is to use an ID scheme that is unique for each camera. This makes things just a single lookup when the application is started. As it stands, the application simply waits for the user to select a camera before providing a live view.

8.4 Custom Layouts

The display layout allows for configurable camera layouts. There is no way for an end user to supply their own layout files, though. All of the display layouts are embedded into the application itself. There is already a layout manager that can be easily leveraged to add more layouts. The actual work here is to define some user directory for layouts or a way for the user to load

a custom layout that the application can remember. This does not address layout creation, however.

Creating a new layout is not easy enough for an end user. They are forced to design the layout in XML and validate that the rows and columns add up properly. It may be a useful feature to provide a user interface for the user to easily create this XML file and have it added to the list of available layouts.

8.5 Camera Synchronization

Camera synchronization is a big piece of the HemView system, but this capability should be extended to support other multi-camera array systems. As it stands, the support is a bit weak as discussed in Section 7.5. It would be valuable to see what can be done to make the synchronization a bit better for these types of arrays and extend it to support more than just firewire or some other technology.

One idea for the HemView system in particular is to pay more attention to the image timestamps. Firewire cameras from PointGrey keep this timestamp when the image is taken and picked up by the host system. These timestamps can be expected of any firewire cameras and could be used to manage the synchronization. The real trick here is that as the number of cameras in the array increases, the likelihood that it uses multiple buses increases. This poses an interesting challenge since the timestamps will not match between the two firewire buses.

8.6 Snapshot

Snapshots are very important for surveillance programs. In many ways, the application already has the ability to do screen shots, but this ability has not been brought up so the user can take advantage of this feature. It is also very limited in that it relies on the camera display to get the snapshot. It might be useful to consider modifying how this snapshot is taken so that specialty cameras can have a special image encoding. As an example, HemView systems may wish to store each camera as its own image. This would allow the system to render all the camera images in a single

OpenGL scene and allow the user to better explore the snapshot.

8.7 Recording and Playback

Similar to snapshot, recording and playback is still needed in the application. It could be implemented at the display level, but this only leaves a recording with a 2D view of the scene. For cameras like HemView, it is imperative that the recording subsystem can handle bringing multiple images together and properly compress them using some video codec. One idea to accomplish this is to treat each image as an independent stream and encode them in separate files. This is an easy approach as existing codecs can easily handle encoding single streams. The second idea is to do a similar operation, but instead store all the encoded streams into a file and manage the file metadata so that the streams can be pulled out and decoded separately.

File playback needs to be supported for these multi-image video formats as well. Ideally, metadata in the video file would allow the system to automatically select the proper display for the video. For example, a HemView video recording would properly load the HemView display plug-in and show the video in an OpenGL scene. The difficulty of doing this will depend on how advanced the encoding scheme is and if there is a good way to manage the content types of the different videos.

8.8 Image Quality Selection

An important option is to allow the user to select the resolution and general image quality for a stream. This would allow the user to control how taxing the application is on the their system. This will also affect the frame rate and general speed of the user interface. In some ways, this feature is covered by Section 8.1, but this is a more general concept supporting the idea of variable quality between recording and live playback. As an example, one may use a higher quality debayer algorithm when taking a snapshot, but then switch to a lower latency solution during live playback.

8.9 Computer Vision

One of the major next steps was to introduce computer vision algorithms to the mix. The general idea is that with the generalization of the data passed between filters, it should be possible to implement filters that provided some kind of computer vision support. As a basic example, motion detection could be implemented that would show itself in the form of a filter plug-in. Downstream filters could take advantage of this motion information and add things to the final display image. The one major problem is that the architecture was not quite built to support this kind of work.

This is one of the major spaces lacking in the framework. The filters in the trees may need to talk to and work with each other to produce the final image. As it stands, the framework does not natively support this functionality. Now, this does not mean that a filter provider could not add support for its filters to work together. As an example, the HemView plug-in provides a filter that synchronizes multiple cameras together. This functionality should be natively supported by the framework, though, to make computer vision filters work together in an easier way.

8.10 Network Camera

One of the future capabilities of the software is to support network cameras such as a HemView system. Just as IP cameras make life much simpler for video users, having the ability to connect to other types of networked cameras is very useful. A secondary benefit of this ability is that a remote video file server could be created and this interface could connect to that for a video feed. The various options are endless and are only limited by the different camera vendors.

CHAPTER 9

CONCLUSIONS

Now that all the performance analysis and architectural discussion is finished, this chapter reflects on the four major goals of the experimental architecture. To review, these goals were to design a robust architecture achieving 25 FPS with a sensor of 5 MP, to create a flexible HemView architecture, to provide a HemView display with proper synchronization, and to support many different kinds of cameras with a single interface. This chapter will look at each of these goals in turn and discuss the success or failure of each.

The first goal was a frame per second goal of 25 FPS with an image sensor of at least 5 MP. This goal was easily reached by the architecture, showing a potential throughput of 83.3 FPS. Even though the final image display was only 7.5 FPS, it was shown that firewire bus bandwidth limitations were holding back the application from showing a better frame rate. For that reason, this goal was easily achieved and showed some of the potential of the architecture.

The second goal was to provide a HemView camera interface that was easily adaptable to different camera subsystems. With the use of the plug-in system and the XML configuration files, this goal was accomplished and demonstrated with the use of a connected IP camera; modifying two lines in the HemView's configuration file was all it took to swap from a firewire camera to an IP camera. Since this modification is simple and straightforward, this demonstrates that the HemView architecture supports interchangeable camera sub-systems.

The third goal was to provide proper synchronization for the cameras used in the HemView camera system. This goal was only half achieved. While cameras on the same bus were properly synchronized in the final display, cameras across the firewire bus boundary were not properly synchronized. The lag was as bad as a few frames and was very noticeable in the final display. The claim for synchronization cannot be made. Therefore, the

architecture failed to reach the synchronization goals.

The final goal of the system was to support varying shapes, sizes, and types of cameras. Thanks to the use of IP cameras from TRENDnet, firewire cameras from PointGrey and the HemView system from VTI, the architecture has been exercised enough to show that it can support a wide range of cameras. As an extension to this point, it is important to note that all of the HemView cameras are supported by this system. This is important because the old software was unable to accomplish this and this ability was not featured in the analysis chapter due to only having one system available. It is, however, important to mention this point as it helps to support the claim that the architecture is robust in its support for different cameras.

Overall, the experimental architecture was a success. The goal of image synchronization is not too far off and is a first next step for future work. All of the other goals were successfully met and a final product was delivered to the HemView producers, VTI. In the end, the architecture proved its flexibility and that it can remain relevant as camera technology progresses.

REFERENCES

[1] "Vision Technology Inc." 2011. [Online]. Available: http://www.focused-mosaics.com/

[2] "OpenCV," 2011. [Online]. Available: http://opencv.willowgarage.com/wiki/

[3] "ZoneMinder," 2011. [Online]. Available: http://www.zoneminder.com/

[4] "ISPY CONNECT," 2011. [Online]. Available: http://www.ispyconnect.com/

[5] "multitude," 2011. [Online]. Available: http://code.google.com/p/multitude/

[6] "sentry360," 2011. [Online]. Available: http://sentry360.com/

[7] "Point Grey." [Online]. Available: http://www.ptgrey.com/

[8] "Ladybug," 2011. [Online]. Available: http://www.ptgrey.com/products/spherical.asp

[9] "Qt," 2011. [Online]. Available: http://qt.nokia.com/products/

[10] "Qt Creator," 2011. [Online]. Available: http://qt.nokia.com/products/developer-tools

[11] "FlyCapture SDK," 2011. [Online]. Available: http://www.ptgrey.com/products/pgrflycapture/

[12] "libdc1394 Homepage," 2011. [Online]. Available: http://damien.douxchamps.net/ieee1394/libdc1394/

[13] "Trendnet," 2011. [Online]. Available: http://www.trendnet.com/?todo=home

[14] "Gpu-z," 2012. [Online]. Available: http://www.techpowerup.com/gpuz/